This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2014.2331682, IEEE Transactions on Emerging Topics in Computing

Towards Consistency of State in MMOGs through Semantically Aware Contention Management

William Blewitt, Matthew Brook, Craig Sharp, Gary Ushaw and Graham Morgan

***Abstract*—This work presents a protocol which utilises semantically aware, transaction-based contention management to reduce rollback and improve consistency in massively multi-user systems. Of particular relevance to Massively Multiplayer Online Games (MMOGs), the proposed system is adaptive and scales with respect to connection latency. The work presents some background to the area of state management and consistency in MMOGs, and their impact on user Quality of Service (QoS). Our solution is then outlined in significant detail, with particular attention paid to the manner in which it maps to the structure of MMOGs. A simulation is demonstrated and its behaviours discussed in-depth in order to support arguments regarding the suitability of the protocol.**

***Index Terms*—Consistency, Distributed Systems, MMOGs.**

I. Introduction

A Massively Multiplayer Online Game (MMOG) is an entertainment-oriented, large-scale and distributed application. Such simulations have been described more generally as *distributed virtual environments* (DVEs) [1], and their implementation represents a significant engineering challenge. An aspect of this challenge relates to the issue of consistency of state across multiple users sharing the same environment and the manner in which that consistency, especially in the context of persistence, is maintained at a level that secures QoS for the player [2, 3]. We define state, in this context, as the set of values associated with all data items stored on the server and updated by the client; we define consistency as agreement between clients regarding state.

Historical approaches to the solution of this problem have generally directed their focus towards either conservative or optimistic synchronization algorithms [4, 5]. In the former case, inconsistencies are avoided through a process of lockstep advance of the simulation's logical clock, ensuring that all clients share a completely uniform game state. This approach is impractical, however, in scenarios with multiple clients because the restrictiveness of such a policy hinders the meeting of real-time requirements of DVEs. In context of optimistic synchronization schemes the generally observed principles are that a client is free to continue to advance its local simulation until an inconsistency is observed whereupon a rollback, or backup [6], is initiated.

Optimistic approaches have enjoyed significant attention [7, 8, 9, 10] due to the fact that their advantage in speed offsets the loss of immersion caused by occasional, latency-induced rollbacks. Through intelligently applied, context-specific design decisions related to the nature of the game experience, QoS can be sustained to the satisfaction of the player [11]. By extension, however, this approach necessitates management of user expectations and places significant constraints upon designers; as such, systems and approaches which limit the instances and magnitude of rollback are of direct relevance to industry.

In the context of a commercial product, a player's in-game view does not necessarily reflect the data which his or her system acts upon; rather, the in-game view represents a trend towards accuracy, providing a layer at which compensation can be applied in a gradual sense. In this way, rather than the player experiencing a rollback event, their game experience tends towards consistency (in the sense of an 'eventually consistent' system) with their system's local database copy, while their system's local database copy tends towards consistency with the server's arbitrating database (in that same sense).

Expanding upon previous work in contention monitoring for distributed systems [12], this work presents a novel approach to consistency control for distributed and shared variable updates within the context of MMOGs. Based upon semantically optimised transaction management, the proposed system facilitates optimistic synchronisation while demonstrably minimising any required compensation. The system is particularly suited to dealing with partially-predictable data access patterns, lending itself towards tasks such as MMOG physics and environmental variable updates.

We begin in Section II with an overview of contemporary practises and research regarding state management and consistency in MMOG architectures. In Section III, our work goes on to outline our proposed approach in explicit terms, describing first the overall system architecture and next the functional protocols that govern its behaviour. We present in Section IV a testing simulation of our architecture and, in Section V, we consider in detail the behaviours observed. Section VI concludes our work with some discussion of the implications of our results, and presents proposed future expansion of the system.

II. Background and Related Work

Online games are by their nature systems built upon optimistic replication of state [13]. In the case of our system we have adopted a narrower focus than is generally

W. Blewitt, M. Brook, C. Sharp, G. Ushaw & G. Morgan are with the School of Computing Science, Newcastle University, Newcastle, NE1 7RU, United Kingdom. Correspondence E-mail: w.f.blewitt@ncl.ac.uk

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2014.2331682, IEEE Transactions on Emerging Topics in Computing

reflected within the literature. Partly, this reflects the fact that the focus of our work has wider application than the area of MMOGs. It also represents an intention on our part to present a solution that forms a specific layer of the application and which is, in and of itself, implementation-agnostic.

In this Section we discuss the background context of our work, with particular relevance to optimistic execution as it is impacted by the nature of MMOGs. We consider an MMOG a rich internet application which maintains local replicas of shared states on the terminals of all inter-related clients; the determination of client inter-relationship, along with other issues revolving around the architecture and behaviour of MMOGs, is discussed in Section II-B. Prior to this, we consider the more general area of optimistic replication, and the place of transaction-based causality management within that field.

A. Optimistic Replication for MMOGs

Optimistic replication schemes provide an eventually consistent guarantee for the replica data served to clients and are desirable for one reason: they are scalable [14]. Eventual consistency, in basic terms, describes a system where all data replicas will become consistent at some point in the future if updates cease. Theoretically, if there exists a window of time long enough where updates are no longer made then clients will be provided with mutually consistent views of the data. It follows that if updates are relatively rare compared to reads, then inconsistency will be present, but for many applications this level of inconsistency is tolerable. For example, search engines and resource management in cloud infrastructures, where scalability is paramount, employ optimistic schemes [15, 16]. Inconsistencies in such application types can be ignored as the overall correctness of the execution is unaffected (e.g., getting slightly different results from a search is typically irrelevant, and clients would never know anyway).

The earliest work in eventually consistent systems was primarily concerned with achieving as much consistency as possible. As all previous protocols strained to achieve full consistency, but were not scalable, the notion was to accept slightly lower consistency to gain scalability. Bayou and IceCube [17, 18] are infrastructures which allow read/write requests to be reordered with the aid of programmer defined relationships in order to increase the processing of requests throughout the entire system. In such approaches inconsistency was not always ignored, but tended to be handled in some deterministic manner. For example, enacting compensation was a common mechanism. Typically, data found to be inconsistent at a client would have to be updated retrospectively (to ensure it is consistent), with the action that enacted the state change either being compensated for or undone and forgotten. If causality was an issue (i.e., subsequent actions may have depended on the compensated/undone action further actions may need to be compensated or undone).

The earlier work in optimistic protocols assumed that causality could be maintained at the semantic layer. Specifically, that application specific semantics could be exploited to identify situations where a compensatory action could be executed instead of forcing the application to rollback execution. Consider, for example, a client that realises that an action could not be achieved 20 steps in its past. A truly causal system without application semantic knowledge would have to rewind all 20 steps and start again. Without semantic knowledge such systems are actually implementing a transactional like approach (where abort, rollback and restart are typical). Therefore, it is the semantic knowledge that brings the scalability to eventually consistent replication protocols, be it either ignoring inconsistencies in heavy read/light write systems or enacting programmer directed compensation in earlier systems.

MMOGs, in essence, implement eventually consistent data guarantees that share many traits with earlier incarnations of optimistic replication protocols. The programmer utilises semantic knowledge to determine if causal infringement or inconsistency matter and what should be done. For example, if a unique inventory item was later found to be in the possession of two players simultaneously then some compensation would be enacting to remove (in a realistic, game flavoured manner) that item from one of the players. Actions carried out that are now considered causally infringed would also be compensated for within the game-play layer, so any consequences unfolded as expected for the player over time. However, this strong semantic causal relationship (a direct reflection of progressive game play) means that MMOGs are more related to transaction like environments for some game play instances.

If transactional like issues arise in MMOGs for certain aspects of game play then an eventually consistent protocol is required that eases the programming burden of handling real-time compensation at the game play layer. In protocol design we strive to generalise as much as possible the requirements of the application and place them within the protocol so as to avoid re-implementation at the application layer. Therefore, the MMOG programmer requires an eventually consistent protocol that can provide the highest throughput while minimising those inconsistencies requiring the greatest compensation. As MMOGs exhibit a read/write equilibrium this is not a trivial task. However, there is one aspect missing of all current optimistic replication schemes that would help in achieving our goal: contention management.

Contention management is simply a mechanism for indicating which transaction should succeed and which should abort. In high throughput systems where strong causal relations are expressed across a system (e.g., linearizability in transaction memory), the type of contention manager chosen has a significant effect on application performance [19]. This is not a consideration in state of the art optimistic schemes (e.g., one search engine client dynamically interacting with another is not applicable). Therefore, if MMOGs are to make use of optimistic replication to ensure their eventual consistency then their transactional like qualities will, one assumes, require contention management to be provided. We have shown with other application

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2014.2331682, IEEE Transactions on Emerging Topics in Computing

domains (eCommerce, High Frequency Trading) [12, 20], that adding contention management to optimistic replication can bring about dramatic improvements in performance. Furthermore, by combining client injection rates with contention management we have shown that overall performance can be increased [21].

MMOGs, fundamentally operate an optimistic replication policy across clients and a central server with strong causal requirements for particular aspects of replica data that are crucial to correctness of gameplay. Therefore, we suggest that the use of a contention manager for those replica data items that are crucial to overall game play correctness should improve performance and lower inconsistencies. As contention managers are application dependent, we need to consider how existing approaches handle consistency of state. This section continues with a more detailed discussion of replica management in current MMOG architectures.

B. Consistency in MMOGs

The ability to control the level of causality is a significant factor in perceived QoS in MMOGs. Observed rollback is more noticeable to the player than adaptive compensation.

The level to which compensation of optimistic execution is noticeable is a function of the magnitude of the inconsistency and the context of the state property which has become inconsistent. That is to say: a variable which has diverged significantly from its optimistically predicted value is more likely to be noticed in compensation than one which has diverged only slightly; a variable to which the player devotes significant attention is more likely to be noticed than one of which he or she is only peripherally aware.

The magnitude of the inconsistency of state is, itself, closely related to the time taken by the client to recognise, or be informed of, the need to adapt the results of its optimistic execution. This relationship can be both direct, meaning that an optimistic protocol which trends away from the actual value produces a result that is on some level proportional to the time it is in effect, and indirect, meaning that other variables within the system have optimistically reacted to the contentious value.

In the latter case, from a QoS perspective, an important factor is the player's reaction to state information which is later proven erroneous, and how to compensate for subsequent events without both breaching suspension of disbelief and/or causing a ripple of further inconsistencies throughout the massively multiplayer environment. As such, a system which reduces the level to which inconsistencies are generated in the initial instance is of direct value to both existing architectures and future MMOG implementations.

We consider in particular the issue of player-observed latency of action as it impacts perceived causality, where the context of an action is related to the contextual importance of its semantic consistency. Claypool and Claypool [22] consider the gamut of online video games in their assessment of precision and deadline. They conclude that while third-person MMOGs include player actions that require either high precision or a narrower deadline for completing an action, they do not require both. Claypool and Claypool also highlight the concept that the impact of latency upon performance is greater when the player is performing precise actions; we extend this assertion to include perceived inconsistencies as relating to experienced latency [23].

Suznjevic et al. [24] also consider the issue of precision against deadline, but solely from the perspective of MMOGs. They observe that even within the context of an MMORPG there are several different play scenarios (which they define as Action Types), each of which has a different requirement in terms of precision and deadline. They argue, for example, that large-scale cooperative play against non-player entities ("Raiding") has a higher precision requirement than player versus player ("PvP") combat, while PvP combat has a tighter deadline requirement than Raiding.

While the shape of the relations between deadline and precision vary between the works of Claypool and Claypool and Suznjevic et al., that variance in itself suggests that optimising contention management even within existing MMOG archetypes requires a fundamental level of semantically suitable dynamism. Furthermore, it reinforces the point that reduction in required compensation of action, which by extension tightens deadline, facilitates not only greater QoS for players of MMOGs styled after existing design paradigms, but empowers alternative design paradigms which that variance in deadline might previously have rendered impracticable.

C. Server Topology in MMOGs

When considering the structure of the system design outlined in Section III, it is important to reflect upon the presumed topology of the application to which such a system would be applied. Particularly, attention should be given to the concepts of sharding massively multiplayer environments, and the nature of *zoning* through the application of distributed back-end servers.

Shards are considered independent instances of the same game world, the application of which is commonly employed in commercial MMOGs as a means of permitting scalability while maintaining QoS [25]. Some successful commercial MMOGs have followed a single-shard model [26], and the system outlined in this paper is functionally independent of the level of sharding pursued by the developer.

Another commonly discussed practise in MMOG development is the process of sub-dividing the computational processing of events in a given shard on the basis of a regional area of interest (AoI), often referred to as zoning [27]. Significant literature has been published on the engineering and structure of such systems [28, 29], and the system outlined in this paper does assume the application of this technique.

D. Contribution of Paper

Our work presents a novel, dynamically optimised approach to contention management in a DVE. Specifically,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2014.2331682, IEEE Transactions on Emerging Topics in Computing

we present a system whereby existing practises within MMOG development could increase QoS through contention management for semantically appropriate data. Furthermore, the performance benefits offered by this approach invite the exploration of novel, consistency-centric mechanics in MMOG design which are often overlooked due to the consistency considerations they impose (e.g. greater integration of physics in MMOGs).

III. Approach

In this section we describe our approach, which is based upon work first presented by Brook et al. [20]. We pay particular attention to client-side and server-side responsibilities with pseudo code provided to describe the activities of each. Algorithm 1 describes the client protocol while Algorithm 2 describes the server. [12] provides a detailed outline of the fashion in which our initial exploration of semantically aware contention management was implemented, though it does so without reference to dynamically adjustable client injection rates. Similarly, it does not cater for application scenarios with changing access trends, which is very much the case in MMOGs. The system is presented both in general terms, as its benefits extend beyond solely the application domain of large-scale DVEs, and in context of the application-specific aspects of system design as relevant to the domain of MMOG implementation.

A. Overview of System Design

Within our system, network clients represent players that are connected to an instance of a game world and share a specific AoI. The instance of the game world, or *shard*, manages a collection of servers, or *sub-shards*, each of which processes activities within a bounded region of responsibility. A sub-shard maintains a correct and consistent database of shared variables for its associated zone. We stress our chosen nomenclature because, in the context of MMOGs, the term 'server' is often used in reference to a shard, with consumers only peripherally aware that a shard is often a complex, distributed system consisting of multiple 'servers'. Figure 1 indicates an idealised system, where load balancing is handled at or about the shard level, and the database under consideration contains information that is solely relevant to the appropriate sub-shard's region of responsibility.

All clients maintain a local replica of their semantically relevant sub-shard's database. A client enacts updates to variables on its local copy of the database based upon its interactions with objects within its zone, facilitating a highly responsive environment for the player. The client utilises a number of logical clocks to facilitate management of execution and rollback. We note that in the domain-specific context, the magnitude of rollback represents the number of actions which require compensation to bring them back in line with actual results, rather than events which require repetition and re-computation; in an MMOG, perceived QoS is better served through compensation than

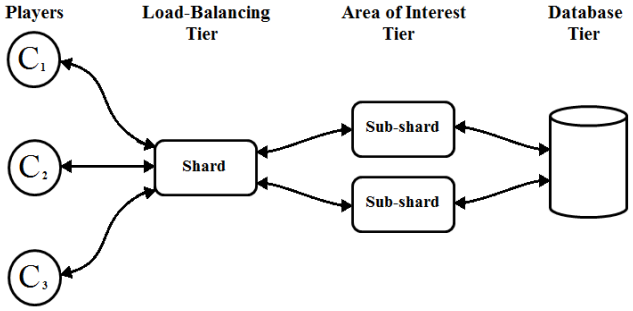

Fig. 1. System Design

implementation rollback. As such, where the term rollback is employed, it should be taken to mean an instruction to compensate those actions which, in another domain, would be rolled back.

The three logical clocks employed by the client to manage causal consistency are:

Client data item clock (*CDI*)
This exists for each data item and identifies the current version of that item's state as stored in the given client's replica. It is used to inform the system when a client's view of said data item is out of date. The value of the CDI is updated incrementally by the client when its associated data element is updated locally, or when a message is received from the server instructing the client that a conflict has occurred (see Algorithm 1, line 6).

Client session clock (*CSC*)
This value is attached to every proposed update transmitted from the client to the server (line 10). When a client is instructed of a rollback $compensation$ condition, this value is updated (line 3). In this fashion, the server is able to ignore subsequently received updates from that client which belong to out of date sessions (i.e., were processed and transmitted before the client was informed of an inconsistency).

Client action clock (*CAC*)
This value is incremented each time a client sends a message to the server (line 11). In this way, the server is able to determine when messages from a client are failing to reach it.

Updates performed locally on the client terminal are propagated to the server in the form of a message. The message contains the updated data item value, the CDI of the data item, the CSC of the client, and the CAC of the client (see lines 8-13). Each client maintains an execution log to facilitate rollback; each time a client sends a message, that message is added to the execution log (line 14). In a commercial implementation, the execution log is periodically pruned as recorded events reach an

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2014.2331682, IEEE Transactions on Emerging Topics in Computing


```
function ClientAlgorithm(Tuple : update)
    Message : resp, req;
    ServerID : sid;
    Client : client;
    Database : D;
    Queue : messages;
1   while messages ≠ ∅ do
2       resp ← Dequeue(messages);
3       client.csc ← resp.csc;
4       client.cac ← resp.cac;
5       Rollback(D, resp);
6       {Update(t, D) | Tuple : t ∈ resp.updates};
7   Checkpoint(D);
8   req.update.data ← update.data;
9   req.update.cdi ← update.cdi;
10  req.csc ← client.csc;
11  req.cac ← (client.cac ← client.cac + 1);
12  req.cid ← client.id;
13  Send(req, sid);
14  Append(client.log, req)
```

Algorithm 1: The Client Algorithm

age at which they are no longer relevant to any feasible compensation.

As with a client, the server maintains three types of logical clock. The clocks facilitate communication of instructions to the client when an update is deemed to be causally inconsistent. These logical clocks are described below:

Session identifier (*SI*)
: This value is the server's view of a client's CSC. The server maintains an SI for each client, and uses this to identify messages from the client which should be ignored (i.e., messages received from an out of date client session). Each time a client is informed of an inconsistency (a compensation event/rollback), the appropriate SI is incremented (see Algorithm 2, line 25).

Action clock (*AC*)
: The AC is the server's view of a client's CAC. As with the SI, the server maintains an AC value for each client, and uses it to identify situations where client messages are lost (line 20). Each time the server honours an action on behalf of a client, the AC associated with that client is set to that client's received CAC (line 31).

Logical clock (*LC*)
: Analogous to the client's CDI, the LC is a value stored at the database with its associated data item; by extension, each data item within the database has its own LC. When the server receives a message from the client attempting to update a given data item, the attached CDI is compared with that data item's LC (line 24); should the LC be greater than that CDI, the update is causally inconsistent (the client has operated upon an outdated version of the data item).

```
function ServerAlgorithm()
    Message : req, resp;
    Database : D;
    Graph : G;
    Map : SI, AC, LC;
    Queue : messages, delta;
15  while true do
16      if messages ≠ ∅ then
17          req ← Dequeue(messages);
18          if req.csc < req.cid ∈ SI then
19              // ignore message;
20          else if req.cid ∈ AC < req.cac − 1 then
21              resp.csc ← req.cid ∈ SI;
22              resp.cac ← req.cid ∈ AC;
23              Send(resp, req.cid);
24          else if
            req.update.cdi ≠ req.update.cdi ∈ LC then
25              resp.csc ← (req.client ∈ SI) + 1;
26              resp.time ← GetVol(G, req);
27              Insert(delta, resp);
28          else
29              Update(req.update.data, D);
30              Update(req.update.cdi, LC);
31              Update(req.cac, req.cid ∈ AC);
32              UpdateGraph(req, G);
            foreach Message : r ∈ delta do
                if r.time > 0 then
33                  r.time ← r.time − 1;
                else if r.time = 0 then
34                  {Append(r.updates, t) | Tuple : t ∈
                    GetUpdates(r.cdi, G)};
35                  Send(r, r.cid);
36                  Remove(delta, r);
            if time to prune then
37              {Prune(e) | Edge : e ∈ G};
function GetVol(Message : m, Graph : G)
38  Vertex : v ← GetVertex(G, m.update.data);
39  Integer : dist ← 0, vol ← 0;
40  while dist < limit do
41      v ← GetMostVolatile(v);
42      vol ← vol + v.vol, dist ← dist + 1;
43  return vol;
function UpdateGraph(Graph : G, Message : m)
44  Vertex : v ← GetVertex(G, m.update.data);
45  Vertex : b ← GetHBV(G, m.cid);
46  Vertex : a ← GetHAV(G, m.cid);
47  {Update(n.vol) | Vertex : n ∈
    GetNeighbours(G, v)};
48  if b = ∅ then b ← a ← v;
    else b ← a, a ← v;
49  if b ≠ a then
50      Edge : e ←GetEdge(G, b, a);
51      if e ≠ ∅ then e.epv ← e.epv + 1;
52      else Insert(G, b, a);
```

Algorithm 2: The Server Algorithms

Upon receipt of a message from a client, the server assesses whether or not it should apply the update to the data item state stored in the database. Initially, it assesses

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2014.2331682, IEEE Transactions on Emerging Topics in Computing

the client message CSC; if the CSC is lower than the server's SI value for that client, the message is ignored (see lines 18-19). In this instance the client has already received a rollback instruction and the server is receiving updates the client attempted before learning of its initial inconsistency.

If the CAC contained in the client's message is at least two greater than the server's AC value for that client, a message has been lost, and the client must be given a rollback instruction on this basis; this is particularly important in the domain of MMOG engineering, as missed messages can be semantically essential to maintenance of perceived causality. We call this specific instruction the *missed message request*, and it contains only an AC and a SI (lines 20-23).

If the LC for the data item to be updated is greater than the CDI in the client's message, then the client has operated upon an out of date version of the data item state. In this case, a message is sent from the server to the client containing an AC, SI and the latest recorded state of the conflicting data item; the client then re-executes using the latest version of the data. We call this the *irreconcilable message request* (lines 24-27).

B. Semantic Contention Management in MMOGs

As with all contention management systems, we exploit a degree of predictability to improve performance. That predictability comes in the form of patterns of sequential data item access which are a function of both application design and player behaviour. Importantly, the client system plays no role in this contention management, only being informed of failed updates and instructions to take compensatory measures; the inability of the client to actively influence contention management is a key consideration in the context of MMOGs, as it aids the prevention of exploitation on the part of unscrupulous players.

Our approach assumes a basic model of patterns inherent within a series of data item interactions. The system is not tuned based upon any advanced prediction algorithm, in an effort to avoid unnecessary domain-dependency, and to emphasise the generalist nature of the approach.

A directed graph represents sequential data item access. The graph contains a vertex for each data item, with interconnecting edges representing the probability of the client's next data item update being applied to the connected vertex. Figure 2 presents an idealised graph with eight vertices, each of which represents a potential physical entity within an MMOG, and the probabilities of progression from one data item update to another, based upon optimistic execution.

It should be noted that the lack of an edge between two vertices does *not* imply that the two data items cannot be updated sequentially, it merely means that such behaviour is not observed as a common pattern within the system in the context of player action. For example, referring once again to figure 2, the lack of an edge between data items DOOR and SWORD does not mean that a client cannot update

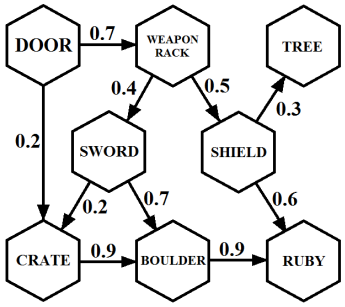

Fig. 2. Directed Graph with Probabilities

SWORD immediately after updating DOOR. It should also be noted that the self-evolving and self-pruning nature of the graph means that there need not be any semantic, design-related connection between two data elements for a link to be autonomously formed between them; as such, the specific nature of the data elements is less important than their existence.

To clarify, we consider single, sequential data item updates in this case, however the model can be extended to reflect multiple, simultaneous chains of data item updates. In such cases, we propose that either the application developer aggregate multiple graphs, or a single graph could be generated which reflects the update patterns of packets of data items (with each vertex representing such a packet). Though maintaining a graph in this matter might appear expensive, we should recall that the number of such shared data items within a given AoI of an MMOG is very small when compared with data repositories utilised in the Cloud, or other modern computing environments.

When informing a client of a failed data item update, the server not only provides the client the latest state of the conflicting data item but, in addition, provides up to date values for neighbouring data items as defined by the directed graph (lines 34-35). The goal of this additional data transfer is that this pre-emptive provision of the most likely candidates for subsequent data item access will lead to reduced inconsistency should the client behave in a predictable fashion.

While maintaining the directed graph, the server also tracks a *volatility measure* for each vertex. Simply put, this variable tracks the 'popularity' of a vertex: how many times a given data item is updated per second. In our proposed system, when the server recognises a failed update it waits a period of time before informing the client of the fact; this length of time is determined by the volatility measure of the data item in question (line 26). The more popular a given data item is, the longer the applied delay (lines 38-43). This aims to prevent situations where multiple clients simultaneously attempt to update the same shared data item; this is a 'back-off' strategy which is commonly employed in shared resource access systems.

While figure 2 shows a static graph (one in which the edges between vertices, and their respective probabilities, are fixed), we employ a dynamic graph which reconfigures based on changes in data access patterns which take place

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2014.2331682, IEEE Transactions on Emerging Topics in Computing

over the course of the application runtime (lines 44-52). We introduce two new values that the server maintains individually for each client:

Happens Before Value (*HBV*)
This indicates the vertex representing the data item that a client last successfully updated.

Happens After Value (*HAV*)
This indicates the vertex representing the data item that a client successfully updated immediately after HBV.

If no edge exists between HBV and HAV then one is established (line 52); if this were left unchecked, however, the graph would soon become fully connected and the system would lose its directional benefit (the number of potential neighbouring data items would increase up to the size of the database itself, less one). Instead, the likelihood of travel down a given edge is informed by an additional value which records the popularity with which an edge between two vertices is travelled:

Edge Popularity Value (*EPV*)
The server maintains an EPV for every edge created between two vertices over the course of the application runtime. The EPV is incrementally increased (line 51) every time a given edge is traversed by any client (as defined by said client's HBV and HAV values).

Periodically, the graph is pruned on the basis of existing EPVs; the least popular edges are removed entirely, and the remaining edges have their EPVs set to 0 (line 37). In this way, the graph is both adaptable and self-regulating, reacting to trends in client data item access (which, itself, is a consequence of player activity within the DVE).

We recognise that the process of reconfiguration incurs a performance cost relative to the number of vertices and edges within the graph. While the decision on how regularly to prune the graph is an application-dependent one, we advocate a strategy that dynamically bases reconfiguration timings upon experienced load. In this case, two factors in particular are important in determining the timing of pruning operations: first, the relative performance cost of the reconfiguration operation; second, the rate of client update activities within the period. One fashion in which to employ this technique in an MMOG is to shadow the database, redirecting to the shadowed copy while the directed graph of the primary database is pruned.

We further observe that the act of reconfiguration offers a window in which the data items themselves can be dynamically altered (changed, introduced, or removed). In the domain-specific case of an MMOG, this allows for periodic, open-world events to occur within the considered AoI.

C. Variation of Client Injection Rate

The contention management scheme as applied on the server side of our system ensures equilibrium of update rates for oft-updated data items within our database. If clients continue to message the server at the same frequency, however, they shall experience increased rollback as their updates are backed off. In the domain of MMOG design, reducing rollback by extension reduces the level to which the client simulation has to compensate for a non-executable action; essentially, a lower rollback leads to a more fluid gaming experience. We determine that this is a desirable property in a system to be applied to MMOG applications.

To that end, we propose the adjustment of injection rates of updates across all clients connected to a given server, the goal being to reach an equilibrium which lessens the magnitude of rollback. Put another way, we wish to optimise the rate at which each individual client updates the server so as to best match the server's ability to successfully process data item updates.

It is suggested that the injection rates of each client be determined through a scheme which defines a rollback threshold. For our purposes we define a rollback threshold to be a number of rollback events that, when observed by a client in response to a failed update, triggers a change in the client's injection rate.

As the client is aware of the magnitude of rollback required upon receipt of a message from the server (by virtue of the client's execution log), a comparison between this magnitude and the pre-defined rollback threshold can be used to inform variable injection rate. This is not a new concept within distributed systems and has been extensively explored [30].

D. Domain-Specific Practical Considerations

Moving between AoIs is an application-dependant process, as is the connection/log-in process itself, and such semantic considerations do not reflect upon the underlying theoretical considerations of our contention management system. We acknowledge that there will likely be additional, implementation-specific communication overhead in a commercial MMOG; our system is concerned solely with updates to shared data elements within the DVE that are subject to real-time influence from multiple connected players.

IV. Testing and Results

In this section we describe the manner in which our proposed system has been evaluated for performance. We first describe the environmental constraints applied to the testing environment, before going on to discuss the tests themselves. We then illustrate our results.

A. Environmental Parameters

Our testing system is implemented through a discrete event simulation built using the `simjava` framework

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2014.2331682, IEEE Transactions on Emerging Topics in Computing

[31]; all results presented in this work are drawn from this simulation. The event simulation was executed on a Core 2 Quad PC running at 2.66GHz with 4GB of RAM and Ubuntu 11.10 was the chosen operating system. The `simjava` framework was selected, in part, due to the authors' familiarity with its functionalities. Additionally, the work presented here extends upon work the authors had already undertaken within the `simjava` framework. The work included could easily be extended into other frameworks with similar functionality. The architectural model as implemented is represented in Figure 1, described in context below:

- The Client tier represents *players* of the game, who are interacting with a number of shared data objects. In context, these might be considered any item or variable within the game which can be influenced by any player and which are represented in-game to all players within a given AoI; we assume that perspective/draw-distance culling, if employed on a graphical level, is not employed in terms of requisite database updates, so as to give a conservative performance estimate.
- Load-Balancing occurs at the Shard level; essentially, the Shard manages connection of the player with the Sub-shard associated with her specific Area of Interest. In a commercial case, this layer would be preceded by a log-in server which would determine the appropriate Shard to connect a player to, assuming a multi-Shard release. This, however, has no impact upon the technique we employ (such servers establish sticky sessions at the beginning of a player's connection, but are generally not a conduit for game data after that sticky session is established) and would serve only to complicate the system design unnecessarily.
- The Area of Interest tier, or Sub-shard, manages data accesses and updates for players active within its bailiwick. It serves as arbiter for both server-side decisions and optimistic player updates to the environment, managing the dissemination of messages informing the player system in case of the former, and acting in accordance with the system presented in Section III in case of the latter. In a commercial scenario, the Sub-shard also communicates with the Shard regarding events which could impact other Sub-shards, and also regarding the passing of players between Sub-shards as they move from one AoI to another.
- The Database tier, in context of our simulation, is the collection of all shared data objects within a specific AoI (meaning those whose values are arbitrated by a specific server). The nature of the data in question is semantic and, as such, a function of the specific game context; for the purposes of our experimentation, each data item simply exists, and is subject to influence from all players active within its associated AoI.

Our testing system employs the techniques discussed in Sections III-A, III-B, and III-C. These techniques are employed to manage a database of 500 such shared data items within the AoI of the simulated Sub-shard. Client data accesses follow the graph in 90% of cases; in 10% of cases, a random data element is accessed that does not follow the graph. The graph itself is pruned at fixed, 30-second intervals. Additionally, testing results are included for a system that only employs the technique of variable client injection rates (Section III-C). This is to facilitate performance comparisons as to the observed benefits of a system employing back-off, relative to one which does not.

In addition to this variance, we consider variable latency requirements, and consequences to our system, within the context of MMOGs. Claypool and Claypool [32] argue that connection latency impacts online gaming experience in a highly semantic fashion; that the type of online game being played is related to the tolerance of the player for latency of action. They go on to define threshold values for playability dependent upon type of online game being played: less than 100msec for first-person games; less than 500msec for third-person games.

Sensible of the growing diversity in MMOG releases, including 'First Person Shooter' games [33], it was deemed sensible to consider the effect of perceived latency upon the system proposed in this work. Drawing inspiration from the work of Claypool and Claypool, and the considerations of Chen et al. [34], we define two latency bands for our experiments. The first, termed *low latency*, is a random distribution of round-trip latency between 10msec and 60msec; the second, *high latency*, is a random distribution of round-trip latency between 50msec and 300msec. This facilitated the drawing of domain-specific inferences during the consideration of results.

Our 'No-Backoff', comparison system always operates at *low latency*.

Aside from latency, a significant consideration in the context of any shared-data system is the number of agents interacting with a given database. Nae et al. [27] present research showing that the population of a given MMOG Shard, for the game RuneScape, varied between 700 and 1,700 concurrently-connected players depending upon the time of day. Similarly, Pittman and GauthierDickey [35] present statistical data showing that, during normal service, concurrently-connected population of a World of Warcraft Realm (Shard) varied between 300 and 1,600 dependant upon the time of day.

Our system concerns itself solely with a specific Sub-shard of a DVE and, as observed by Nae et al., the player capacities of Sub-shards directly influence the hosting cost of a Shard and, consequently, the ongoing running costs of an MMOG. In light of this commercial consideration, taken in view of the estimated population figures above, our concurrently connected clients per Sub-shard vary from 50 players to 250 players. This enables us to draw performance comparisons based on Sub-shard occupancy, and consider the consequences of higher populations upon our system performance.

The last variable we consider in our testing relates to Variable Client Injection Rate as discussed in Section III-C. The pre-defined rollback (or compensation) threshold used to trigger variance of injection rate is directly controllable

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2014.2331682, IEEE Transactions on Emerging Topics in Computing

by the application programmer. In the experiments performed as part of this work, two values for that rollback threshold were employed: 30 rollback events, and 60 rollback events.

B. Results

We present our results in terms of four semantically important values, varied according to the number of concurrently connected players interacting with the simulated Sub-shard. We present these values, and their semantic relevance, below:

Number of Unsuccessful Updates
This variable signifies the total number of unsuccessful attempts by player systems to update the database over the course of the simulation. Semantically, this is a consequence of either a missed message from the player's system, or an attempted update based on old and inaccurate data. The lower this variable, the lower the requisite message traffic of the system.

Level of Inconsistency
This variable identifies the number of inconsistent data items experienced by player systems during the simulation. The lower this value, the more consistently our DVE is experienced by the players interacting with its data objects and, consequently, the greater the level of shared multiplayer experience.

Average Compensatable Actions
This variable signifies the average number of compensatable actions (commonly referred to as rollback events) experienced by a player system in the event of an unsuccessful update. The lower this value, the greater the resemblance between what an individual player sees on her screen and the data upon which her system is operating. Put another way, a lower number of compensatable actions leads to a more consistent play experience.

Update Throughput
This variable marks the number of successful updates processed by the system, per second. The higher this value, the greater the real-terms level of interactivity between the players and the DVE, and the more fluid the environmental simulation.

Table I presents our recorded number of unsuccessful updates while running our simulated system with a rollback threshold of 30. Figure 3 presents this information in a graphical form.

Tables II, III, and IV show our recorded results for the remaining three key variables: consistency, average compensation, and throughput, respectively. These results were similarly obtained using a rollback threshold of 30.

TABLE I
UNSUCCESSFUL UPDATES (THRESHOLD = 30)

players	Backoff Low Latency	Backoff High Latency	No Backoff
50	14,110	17,208	18,213
100	31,777	38,873	59,336
150	81,649	82,308	124,241
200	134,926	135,059	211,470
250	190,152	197,727	325,659

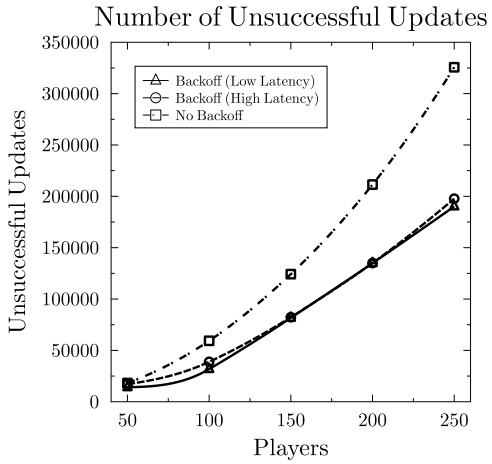

Fig. 3. Unsuccessful Updates (Threshold = 30)

The information present in these tables is presented in graphical format as Figures 4, 5, and 6, respectively, to facilitate clearer comparisons in Section V.

Tables V through VIII demonstrate the results obtained for our four key variables while employing a rollback threshold of 60. Figures 7 through 10 provide this information graphically.

TABLE II
LEVEL OF INCONSISTENCY (THRESHOLD = 30)

players	Backoff Low Latency	Backoff High Latency	No Backoff
50	7.23903	7.05988	8.05988
100	11.30336	11.40118	14.15931
150	12.88053	14.74642	21.83130
200	14.29741	17.55748	27.51134
250	15.24864	17.73830	31.79668

TABLE III
AVERAGE COMPENSATABLE ACTIONS (THRESHOLD = 30)

players	Backoff Low Latency	Backoff High Latency	No Backoff
50	18.49525	19.88367	20.45264
100	19.45049	20.44432	21.40121
150	20.54666	20.84066	22.24104
200	21.45056	21.63286	22.24104
250	21.95747	22.07691	22.72951

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2014.2331682, IEEE Transactions on Emerging Topics in Computing

TABLE IV
MESSAGE THROUGHPUT (THRESHOLD = 30)

players	Backoff Low Latency	Backoff High Latency	No Backoff
50	141.0000	81.8961	64.6313
100	188.5698	113.2227	94.72603
150	169.2212	117.8728	103.6616
200	153.8062	118.8644	106.3973
250	137.5643	121.2718	106.5556

TABLE V
UNSUCCESSFUL UPDATES (THRESHOLD = 60)

players	Backoff Low Latency	Backoff High Latency	No Backoff
50	15,142	15,601	17,378
100	39,104	36,497	38,485
150	81,589	63,369	85,747
200	139,094	105,973	150,452
250	213,876	161,566	232,004

TABLE VI
LEVEL OF INCONSISTENCY (THRESHOLD = 60)

players	Backoff Low Latency	Backoff High Latency	No Backoff
50	6.94729	4.94100	5.68173
100	12.22845	9.05956	10.34691
150	13.19351	11.92226	13.66700
200	14.69107	13.23717	18.59400
250	16.56442	14.94711	23.70323

TABLE VII
AVERAGE COMPENSATABLE ACTIONS (THRESHOLD = 60)

players	Backoff Low Latency	Backoff High Latency	No Backoff
50	37.96130	38.10633	41.22563
100	39.60903	40.53590	43.65240
150	40.63091	40.85543	43.67210
200	41.12305	41.03317	43.24560
250	41.69593	41.74967	43.57650

TABLE VIII
MESSAGE THROUGHPUT (THRESHOLD = 60)

players	Backoff Low Latency	Backoff High Latency	No Backoff
50	123.5276	79.3590	53.6135
100	176.6631	85.8646	72.6054
150	162.9825	102.3581	82.7112
200	148.2668	106.6025	88.4306
250	133.5843	102.8369	93.2112

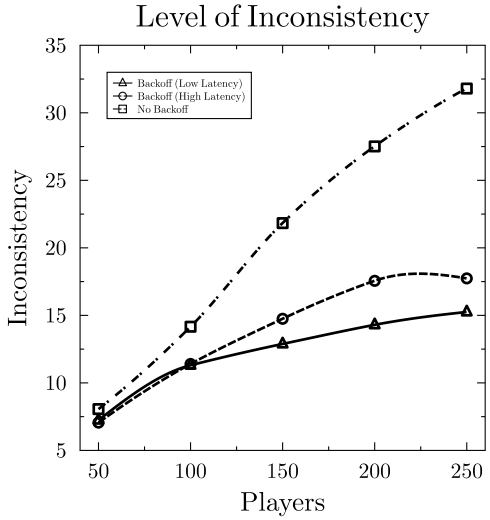

Fig. 4. Level of Inconsistency (Threshold = 30)

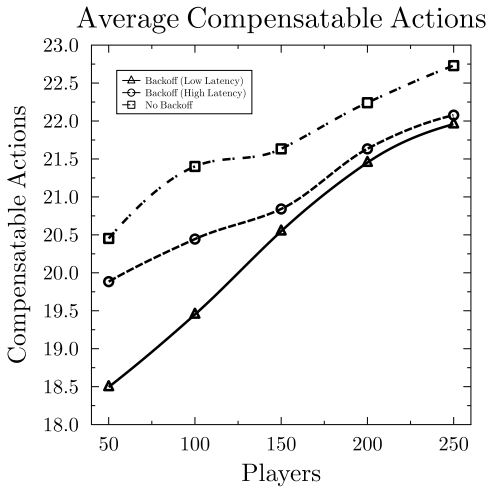

Fig. 5. Average Compensatable Actions (Threshold = 30)

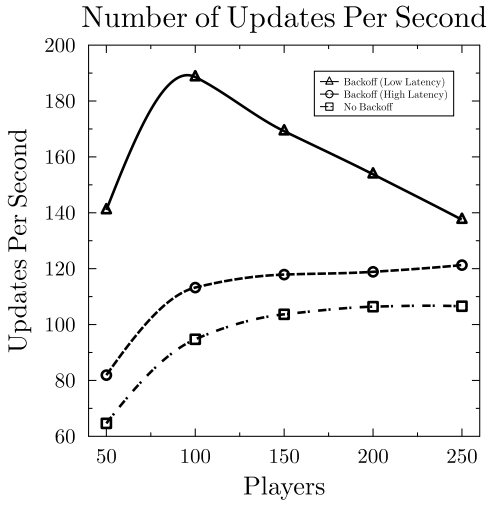

Fig. 6. Message Throughput (Threshold = 30)

V. ANALYSIS

In this Section we consider the implications of the experimental results provided in Section IV in the context of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2014.2331682, IEEE Transactions on Emerging Topics in Computing

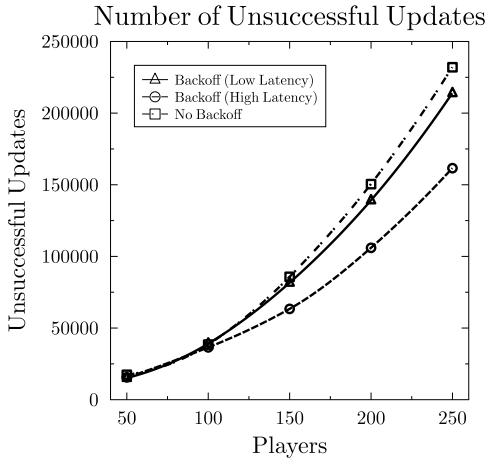

Fig. 7. Unsuccessful Updates (Threshold = 60)

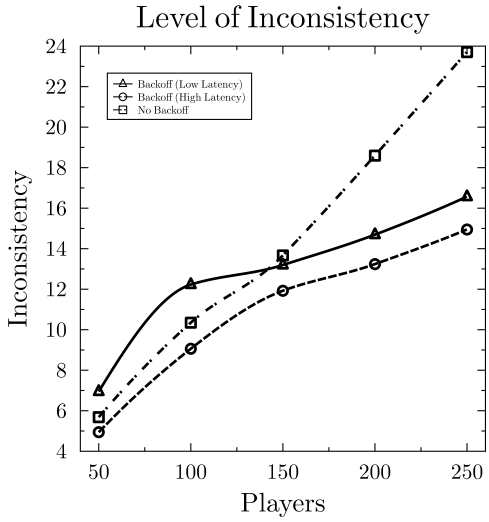

Fig. 8. Level of Inconsistency (Threshold = 60)

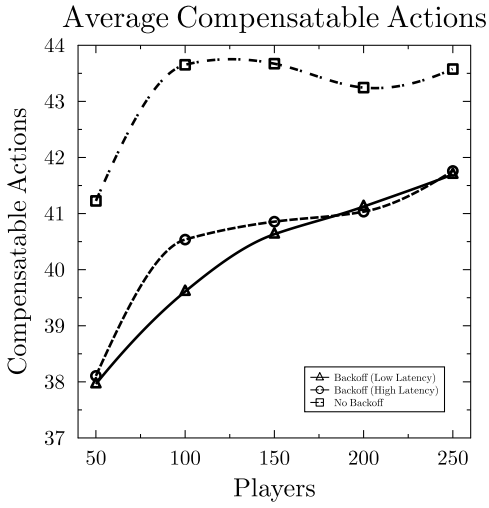

Fig. 9. Average Compensatable Actions (Threshold = 60)

applicability of our approach to the design and development of MMOGs. We have discussed player QoS extensively throughout this work and, in context, reduction in required

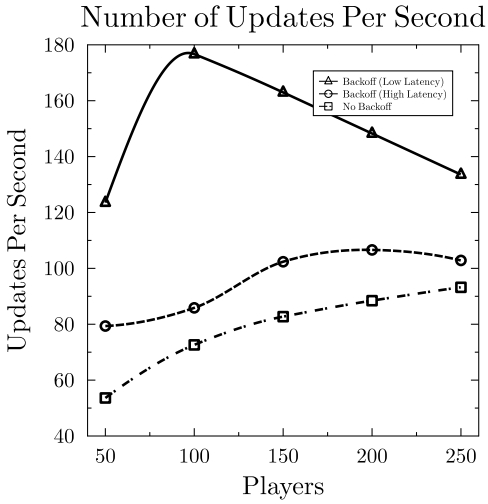

Fig. 10. Message Throughput (Threshold = 60)

compensation facilitates QoS improvement in two ways. Firstly, that in complex and reflex-based MMO gameplay, world view is more consistent from one fraction of a second to the next; this is particularly relevant in player-versus-player content, or in MMOGs which place great emphasis on spatial awareness (such as Carbine's *WildStar Online*). Secondly, it encourages developers to explore mechanics which are traditionally overlooked in MMOG development as a function of world-view consistency issues.

We primarily consider relative performance observed in our testing environment, before providing some consideration of the semantic impacts of rollback threshold variation.

A. Relative Performance

Let us consider the results shown in Table I and Figure 3: Number of Unsuccessful Updates while employing a rollback threshold of 30. Irrespective of the number of players acting within our simulated Sub-shard, the application of backoff reduces the number of unsuccessful updates.

This result in and of itself is unsurprising, though the magnitude of performance improvement warrants some attention. In a low latency environment, with 100 players connected to a Sub-shard, the application of backoff provides a 46% reduction in unsuccessful updates when compared to a system that solely employs dynamic injection. In an environment of 150 players or more, irrespective of experienced latency, the application of backoff improves performance in the context of this variable by over 33%. We note also that a tendency towards linear scalability with respect to number of connected players is observed in the application of backoff, irrespective of latency.

The performance benefit at the higher threshold value of 60, as shown in Table V and Figure 7 is significantly less pronounced. Higher latency (which inherently reduces the number of attempted updates per second) plays a larger role than backoff in determining the total number of unsuccessful updates. In the high threshold environment of our testing system, the application of backoff in a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2014.2331682, IEEE Transactions on Emerging Topics in Computing

low latency environment provides a maximum performance improvement of 12.9%, assuming that only 50 players are connected to the simulated Sub-shard. By contrast, a high latency environment shows an improvement of 26% over a low latency system without backoff, if 150 players are connected.

Let us consider the results shown in Table II, which indicate the average inconsistency of world-view across all players connected to our simulated Sub-shard while employing a rollback threshold of 30. Greater consistency of world-view is a key element of online game design, and a core engineering challenge in the development of massively multiplayer online games.

The application of backoff to this system, in addition to variable injection rate, shows significant improvements in consistency. Assuming 150 players interacting on our simulated Sub-shard, in a low latency environment, their world-view includes 41% fewer inconsistencies; even in a high latency environment, it contains 32% fewer inconsistencies than are present in a low latency, no-backoff system.

Again, as threshold is increased to 60, the performance benefits are negatively impacted; let us consider the results shown in Table VI. Indeed, in a low latency environment with 100 players interacting with our simulated Sub-shard, there is a performance reduction of over 18%. The most notable positive percentage change in performance when adopting a higher rollback threshold is observed in high latency, low Sub-shard occupancy scenarios.

We now consider the results shown in Table III and Figure 5: the average number of compensatable actions processed in the event of an unsuccessful update, while employing a rollback threshold of 30. The lower this value, the less pronounced any actions undertaken by the player system, in order to bring player world-view into line with the Sub-shard's 'true' world-view, become. This facilitates a more consistent gaming experience on the part of a player.

In a low latency environment, with 100 players interacting with our simulated Sub-shard, the application of backoff provides a reduction slightly in excess of 9% in the average number of compensatory actions required in the event of an unsuccessful attempt to update the Sub-shard's database. A performance improvement of 4.5% over the low latency, no-backoff case is observed even in a high latency environment.

In contrast with other variables, the application of a higher threshold provides mild, relative performance improvements in the case of compensatable actions (as shown in Table VII). Assuming 100 players connected to the Sub-shard in a low latency environment, performance benefits improve to 9.25%. In the high latency case, relative performance improvement increases from 4.5% to over 7%.

Let us consider our fourth variable, update message throughput, as shown in Tables IV and VIII. In context, the rate of successful updates determines the timely interactivity of the environment; the higher the rate of successful updates, the more fluid the in-game activity.

When employed in context of our lower rollback threshold, backoff shows significant performance gains in terms of raw, successful database updates. The relative throughput increase is most pronounced in low latency, low occupancy scenarios; in a low-latency environment, with 50 players connected to our simulated Sub-shard, successful update throughput increases by almost 120%. In a more commercially realistic scenario of 100 players connected to our Sub-shard in a low-latency environment, successful update throughput doubles.

In a high latency environment, performance benefits are still significant, though less pronounced. In a similar case, with 100 players connected to our simulated Sub-shard, high latency message throughput increases by 20%. It should be noted, when considering this result, that message update rate is directly, and negatively, impacted by higher latency; it should further be noted that the performance gains observed in successful update throughput are made over a non-backoff system operating at *low latency*.

In the high threshold environment, improvements to successful update throughput at low latency are even more pronounced. Assuming 100 players connected to the Sub-shard, there is a performance improvement of over 140% when compared to the no-backoff case. In the case of a high latency environment, the performance benefits at 100 players are slightly lower than is the case in the low threshold environment, but at a lower occupancy (50 players), performance improvement increases to almost 50% over a low-latency, no-backoff system.

We note in our results data that, at 50 players, absolute numbers of inconsistent updates do not necessarily correlate with average levels of inconsistency viewed globally throughout the simulation. In part, this is a function of a difference between the two metrics; number of failed updates indicates cases where a client attempts to update the server and is instructed to compensate and recompute, while average level of inconsistency is the number of data items duplicated client-side which exist in error. Our future work will include further exploration of this anomaly and its relationship to rollback threshold and latency.

B. Comparison With Respect to Threshold

Our experimentation suggests that, in general terms, the lower rollback threshold of 30 outperforms the higher threshold value of 60. Despite this, there are semantic scenarios where a developer might consider the application of a higher threshold advantageous. In a game environment where individual fluidity of player experience is more important than globally consistent world view, the application of a higher threshold level makes some sense; it leads to marked improvements in both compensatory event reduction and successful update throughput.

By contrast, the results of our experimentation suggest that a MMOG environment that relies heavily upon uniformity of world-view, or which concerns itself more with absolute compensatory requirement rather than relative compensatory requirement, would be better served by opting for a lower rollback threshold.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2014.2331682, IEEE Transactions on Emerging Topics in Computing

VI. Conclusion

In this work we have described an approach for the management of shared data objects within MMOG environments. The system presented facilitates automated self-tuning, combining semantic contention management with variable injection rate to optimise and improve four measures of online game performance. The system presented can adapt to varying data object access trends as caused by alterations within the game world, and engineering solutions have been proposed to enable this real-time optimisation to be undertaken at runtime.

As with many practises currently employed in MMOG consistency management, our system is rooted in optimistic replication. As such, our system has the property of eventual consistency which, in the context of a DVE with constantly changing and updating state, is near-optimal within the limitations of existing hardware and network architectures. The work presented is an extension of earlier work [36, 12, 20, 21] and provides that work with both runtime versatility and a deeper context.

Our experimentation demonstrates the performance gains made feasible by this approach, and we argue that these performance gains offer greater freedom of design to MMOG developers. The results of our experimentation, when taken in view of the work of Nae et al. [27], also have positive financial implications in the context of decreasing the number of Sub-shards required to maintain existing QoS, with respect to a no-backoff system.

The future work surrounding this research will take two forms. Firstly, we intend to investigate the implications of fault-tolerance in terms of performance overhead. Secondly, we intend to extend the system beyond the arena of MMOGs into other forms of online gaming which share optimistic replication characteristics.

References

[1] G. Morgan and K. Storey, "Scalable collision detection for massively multiplayer online games," in *Proceedings of the 19th International Conference on Advanced Information Networking and Applications, AINA 2005*, vol. 1, 2005, pp. 873–878.

[2] C. Diot and L. Gautier, "A distributed architecture for multiplayer interactive applications on the internet," *IEEE Network*, vol. 13, no. 4, pp. 6–15, 1999.

[3] K.-T. Chen, C.-Y. Huang, P. Huang, and C.-L. Lei, "An empirical evaluation of tcp performance in online games," in *Proceedings of the 2006 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, 2006.

[4] R. M. Fujimoto, "Parallel and distributed simulation," in *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future-Volume 1*. ACM, 1999, pp. 122–131.

[5] X.-B. Shi, F. Liu, L. Du, X.-H. Zhou, and Y.-S. Xing, "An event correlation synchronization algorithm for mmog," in *Proceedings of the 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, SNPD 2007*, 2007, pp. 746–751.

[6] K. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[7] S. Zhou and Q. Hu, "The research of improved algorithm of time warp based on distributed network architecture," in *Proceedings of the 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC)*, 2011, pp. 3534–3537.

[8] S.-J. Kim, F. Kuester, and K. K. Kim, "A global timestamp-based approach to enhanced data consistency and fairness in collaborative virtual environments," *Multimedia systems*, vol. 10, no. 3, pp. 220–229, 2005.

[9] S. Ferretti and M. Roccetti, "Fast delivery of game events with an optimistic synchronization mechanism in massive multiplayer online games," in *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology, ACE '05*, 2005.

[10] S. Ferretti, M. Roccetti, and C. E. Palazzi, "An optimistic obsolescence-based approach to event synchronization for massively multiplayer online games," *International Journal of Computers and Applications*, vol. 29, no. 1, pp. 33–43, 2007.

[11] R. Prodan and V. Nae, "Prediction-based real-time resource provisioning for massively multiplayer online games," *Future Generation Computer Systems*, vol. 25, no. 7, pp. 785–793, 2009.

[12] Y. Abushnagh, M. Brook, C. Sharp, G. Ushaw, and G. Morgan, "Liana: A framework that utilizes causality to schedule contention management across networked systems," *Lecture Notes in Computer Science*, vol. 7566, pp. 871–878, 2012.

[13] M. Cajada, P. Ferreira, and L. Veiga, "Adaptive consistency for replicated state in real-time-strategy multiplayer games," in *Proceedings of the 11th International Workshop on Adaptive and Reflective Middleware (ARM '12)*. ACM, 2012.

[14] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[15] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating System Review*, vol. 44, no. 2, pp. 35–40, 2010.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles. SOSP' 07.*, 2007.

[17] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles. SOSP '95.*, 1995, pp. 172–182.

[18] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel, "The icecube approach to the reconciliation of divergent replicas," in *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing. PODC '01.*, 2001, pp. 210–218.

[19] I. William N. Scherer and M. L. Scott, "Contention management in dynamic software transactional memory," in *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[20] M. Brook, C. Sharp, and G. Morgan, "Semantically aware contention management for distributed applications," in *Proceedings of the 13th International IFIP Conference on Distributed Applications and Interoperable Systems. DAIS 2013.*, 2013.

[21] M. Brook, C. Sharp, G. Ushaw, W. Blewitt, and G. Morgan, "Volatility management of high frequency trading environments," in *Proceedings of the 15th IEEE Conference on Business Informatics. CBI 2013.*, 2013.

[22] M. Claypool and K. Claypool, "Latency can kill: Precision and deadline in online games," in *Proceedings of the 1st ACM Multimedia Systems Conference (MMSys)*, 2010, pp. 215–222.

[23] G. Morgan, *Social Networking and the Web*. Academic Press, Inc., 2009, ch. Chapter 3 Highly Interactive and Scalable Online Worlds, pp. 75–120.

[24] M. Suznjevic, O. Dobrijevic, and M. Matijasevic, "Mmorpg player actions: Network performance, session patterns and latency requirements analysis," *Multimedia Tools and Applications*, vol. 45, pp. 191–214, 2009.

[25] M. V. Salles, T. Cao, B. Sowell, A. Demers, J. Gehrke, C. Koch, and W. White, "An evaluation of checkpoint recovery for massively multiplayer online games," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 1258–1269, August 2009.

[26] B. Peers, "Making faces: Eve online's new portrait rendering," in *ACM SIGGRAPH 2011 Talks*, 2011.

[27] V. Nae, R. Prodan, and T. Fahringer, "Cost-efficient hosting and load balancing of massively multiplayer online games," in *Proceedings of the 11th IEEE/ACM International Conference on Grid Computing (GRID)*, 2010, pp. 9–16.

[28] W. Cai, P. Xavier, S. J. Turner, and B.-S. Lee, "A scalable architecture for supporting interactive games on the internet," in *Proceedings*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2014.2331682, IEEE Transactions on Emerging Topics in Computing

of the 16th Workshop on Parallel and Distributed Sumulation. PADS '02., 2002, pp. 60–67.

[29] E. Cheslack-Postava, T. Azim, B. F. T. Mistree, D. R. Horn, J. Terrace, P. Levis, and M. J. Freedman, "A scalable server for 3d metaverses," in *Proceedings of the 2012 USENIX Annual Technical Conference. USENIX ATC-12.*, 2012.

[30] G. J. Nutt and D. L. Bayer, "Performance of csma/cd networks under combined voice and data loads," *IEEE Transactions on Communications*, vol. 30, no. 1, pp. 6–11, January 1982.

[31] F. Howell and R. McNab, "`simjava`: A discrete event simulation library for java," in *Proceedings of the International Conference on Web-Based Modelling and Simulation*, 1998, pp. 51–56.

[32] M. Claypool and K. Claypool, "Latency and player actions in online games," *Communications of the ACM*, vol. 49, pp. 40–45, 2006.

[33] I. Barri, J. Ruis, C. Roig, and F. Giné, "Dealing with heterogeneity for mapping mmofps in distributed systems," *Lecture Notes in Computer Science*, vol. 6586, pp. 51–61, 2011.

[34] K.-T. Chen, P. Huang, and C.-L. Lei, "How sensitive are online gamers to network quality?" *Communications of the ACM*, vol. 49, pp. 34–38, 2006.

[35] D. Pittman and C. GauthierDickey, "A measurement study of virtual populations in massively multiplayer online games," in *Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support for Games, NetGames '07*, 2007.

[36] G. Morgan, F. Lu, and K. Storey, "Interest management middleware for networked games," in *Proceedings of the 2005 Symposium on Interacive 3D Graphics and Games. I3D '05.*, 2005.

William Blewitt obtained a B.Sc degree in Physics with Space Science and Technology at the University of Leicester in 2004, before earning a M.Sc degree in Computational Intelligence and Robotics at De Montfort University in 2006. During his PhD in Computer Science at De Montfort University, he researched computationally inexpensive emotion modelling for AI agents before studying a M.Sc in Computer Game Engineering at Newcastle University. He joined the game technology research group at Newcastle University in 2012, where his research interests include modelling believable agents in real-time systems, heterogeneous computing solutions for AI, and optimisations of general purpose GPU computing for commercial application.

Matthew Brook received his B.Sc (Hons) in Computing Science from Newcastle University in 2008. In 2009 he received his M.Sc in System Design for Internet Applications (now known as Internet Technologies and Enterprise Computing) from Newcastle University. He completed a PhD at Newcastle in 2014 in the area of concurrency control for distributed systems.

Craig Sharp gained a B.Sc in Computing Science (Distributed Systems) in 2008 followed by a PhD in 2013 at Newcastle University. His PhD was focused on the area of Software Transactional Memory and Highly Parallel Computing. He subsequently joined the game technology research group at Newcastle University on the Limbs Alive project which monitors the performance of stroke patients in the context of game play. His research interests include concurrency control and game engine design.

Gary Ushaw received the B.Sc degree in Electronics from the University of Manchester Institute of Science and Technology (UMIST) in 1987, and the PhD in Signal Processing from the University of Edinburgh in 1995. From 1987 to 1991 he worked as an electrical engineer on large scale public projects for communications and control, working with CEGB, British Rail, and the combined electricity generating boards of India. From 1995 to 2011, he worked in the games industry as software engineer and later as engineering manager, focusing on high-end console gaming with publishers including Ubisoft, Sony, Rockstar, BBC and Atari. In 2011 he joined the teaching staff at Newcastle University School of Computing Science, concentrating on video game engineering, rehabilitative gaming, and multicore systems.

Graham Morgan gained his PhD in 1999 and spent the time since studying a variety of areas in computing. With a research background in systems, Graham has published many articles on the engineering challenges related to video game development. This has included collision detection, online gaming, physics engines, graphics, AI and multi-core exploitation techniques.